Exploiting Parallelism in Coalgebraic Logic Programming

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 303 (2014) 121–148

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Exploiting Parallelism in Coalgebraic Logic Programming

Ekaterina Komendantskaya[a,1,4] Martin Schmidt[b,2]
Jónathan Heras[a,3,4]

[a] *School of Computing*
University of Dundee
UK

[b] *Institute of Cognitive Science*
University of Osnabrück
Germany

Abstract

We present a parallel implementation of Coalgebraic Logic Programming (CoALP) in the programming language Go. CoALP was initially introduced to reflect coalgebraic semantics of logic programming, with coalgebraic derivation algorithm featuring both corecursion and parallelism. Here, we discuss how the coalgebraic semantics influenced our parallel implementation of logic programming.

Keywords: Coinduction, Corecursion, Guardedness, Parallelism, GoLang.

1 Introduction

In the standard formulations of Logic Programming (LP), such as in Lloyd's book [19], a first-order logic program P consists of a finite set of clauses of the form $A \leftarrow A_1, \ldots, A_n$, where A and the A_i's are atomic first-order formulae, typically containing free variables, and where $A_1, \ldots, A_n$ is understood to mean the conjunction of the A_i's: note that n may be 0.

SLD-resolution, which is a central algorithm for LP, takes a goal G, typically written as $\leftarrow B_1, \ldots, B_n$, where the list of B_i's is again understood to mean a conjunction of atomic formulae, typically containing free variables, and constructs a proof for an instantiation of G from substitution instances of the clauses in P [19].

[1] Email: katya@computing.dundee.ac.uk
[2] Email: martisch@uos.de
[3] Email: jonathanheras@computing.dundee.ac.uk
[4] The work was supported by EPSRC grants EP/J014222/1 and EP/K031864/1.

http://dx.doi.org/10.1016/j.entcs.2014.02.007

The algorithm uses Horn-clause logic, with variable substitution determined by most general unifiers to make a selected atom in G agree with the head of a clause in P, then proceeding inductively.

Although the operational semantics of LP was initially given by the SLD-resolution algorithm, it was later reformulated in SOS style in [1, 3, 5, 6, 8], and in terms of algebraic (fibrational) semantics in [1, 5, 13, 17]. Logic programs resemble, and indeed induce, transition systems or rewrite systems, hence coalgebras. That fact has been used to study their operational semantics, e.g., in [3, 6]. Finally, the coalgebraic (fibrational) semantics of LP was introduced in [4, 14–17]. The main constructions and results of [14–17] will be explained in Sections 3.1 and 4.1.

When studying the coalgebraic (structural operational) semantics of LP [14–17], we noticed that some constructs of it suggest properties alien to the standard algorithm of SLD-resolution [19]; namely, parallelism and corecursion. This paper will only focus on parallelism, but see [17] for a careful discussion of the relation between the two issues. In particular, [14] first noticed the relation of the coalgebraic semantics to parallel LP in the variable-free case [9], as Section 3.1 explains. However, extending those results to the first-order case with the fibrational coalgebraic semantics [16, 17] again exposed novel constructions, this time alien to the existing models of LP parallelism [10]. The "fibers" present in it suggested restriction of the unification algorithm standardly incorporated in SLD- and and-or-parallel derivations [9] to term-matching; see Section 4.1. This inspired us to introduce a new (parallel and corecursive) derivation algorithm of *CoAlgebraic LP (CoALP)* (see Section 5.1). The algorithm was shown sound and complete relative to the coalgebraic semantics [15, 17].

The original contribution of this paper is parallel implementation of CoALP in the language Go [21]. Go is a strongly typed and compiled programming language. It provides an easy built-in way to use high level constructs to implement parallelism in the form of *goroutines* and channels to communicate between them; this model for providing high-level linguistic support for concurrency comes from Hoare's Communicating Sequential Processes [11]. Go has an easy to set up and use tool-chain and allows for rapid prototyping with its fast compile times, array bounds checking and automatic memory management. In addition, it allows low level programming and produces fast binaries.

Here, we present a careful study of the influence of the constructs arising in the coalgebraic fibrational semantics [15–17] on
(a) CoALP's parallel derivation algorithms;
(b) the Go implementation of CoALP.
Thus, apart from achieving the goal of introducing CoALP's implementation, this paper will serve as an exercise in applying coalgebra in programming languages.

The rest of the paper is structured as follows. Section 2 is a background section, and it introduces various existing (sequential and parallel) derivation algorithms for LP. The rest of the sections follow a common pattern: each splits into four subsections, such that the first subsection studies some constructions arising in the coalgebraic semantics [16, 17], the second subsection shows how those constructions

transform into a parallel algorithm in CoALP; the third subsection explains their implementation in Go; and the last "case study" subsection tests the efficiency of their parallel implementation. Note the emphasis on describing the "constructive" fragments of the semantics, that is, fragments that give rise to concrete algorithms and computations. In this way, Section 3 considers propositional (variable-free) version of CoALP and related Datalog language. Section 4 focuses on CoALP's fibrational semantics and its impact on parallelism. Section 5 discusses semantics and parallelisation of full first-order fragment of CoALP. To reinforce the trend of tracing the constructive influence of the coalgebraic semantics on implementation of CoALP, we recover, where possible, a constructive reformulations of completeness results of [15, 17]; and mark them as "Constructive Completeness" theorems/lemmas. Where a constructive version is impossible, we discuss the reasons. Finally, in Section 6, we conclude the paper.

The Go implementation of CoALP, as well as all the examples and benchmarks presented throughout the paper can be downloaded from [18].

2 Background: Logic Programs and SLD-derivations

We first recall some basic definitions from [19], and then proceed with discussion of parallel SLD-derivations.

A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with a fixed *arity*. The arity of a function symbol is a natural number indicating the number of its arguments. Nullary (0-ary) function symbols are allowed: these are called *constants*. Given a countably infinite set Var of variables, the set $Ter(\Sigma)$ of *terms* over Σ is defined inductively: $x \in Ter(\Sigma)$ for every $x \in Var$; and, if f is an n-ary function symbol ($n \geq 0$) and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $f(t_1, \ldots, t_n) \in Ter(\Sigma)$. Variables will be denoted x, y, z, sometimes with indices $x_1, x_2, x_3, \ldots$. A *substitution* is a map $\theta : Ter(\Sigma) \to Ter(\Sigma)$ which satisfies $\theta(f(t_1, \ldots, t_n)) \equiv f(\theta(t_1), \ldots, \theta(t_n))$ for every n-ary function symbol f.

We define an *alphabet* to consist of a signature Σ, the set Var, and a set of *predicate symbols* $P, P_1, P_2, \ldots$ each assigned an arity. Let P be a predicate symbol of arity n and $t_1, \ldots, t_n$ be terms. Then $P(t_1, \ldots, t_n)$ is a *formula* (also called an atomic formula or an *atom*). The *first-order language* $\mathcal{L}$ given by an alphabet consists of the set of all formulae constructed from the symbols of the alphabet.

Given a substitution θ and an atom A, we write $A\theta$ for the atom given by applying the substitution θ to the variables appearing in A. Moreover, given a substitution θ and a list of atoms $(A_1, \ldots, A_k)$, we write $(A_1, \ldots, A_k)\theta$ for the simultaneous substitution of θ in each A_m.

Given a first-order language $\mathcal{L}$, a *logic program* consists of a finite set of clauses of the form $A \leftarrow A_1, \ldots, A_n$, where $A, A_1, \ldots, A_n$ ($n \geq 0$) are atoms. The atom A is called the *head* of a clause, and $A_1, \ldots, A_n$ is called its *body*. Clauses with empty bodies are called *unit clauses*. We call a term, a formula, or a clause *ground*, if it does not contain variables.

Example 2.1 [`BinaryTree`] The definition `btree` describes a set of binary trees

whose nodes are bits.

```
1. bit(0).
2. bit(1).
3. btree(empty).
4. btree(tree(L,X,R)) :- btree(L), bit(X), btree(R).
```

A *goal* is given by $\leftarrow B_1, \dots B_n$, where $B_1, \dots B_n$ $(n \geq 0)$ are atoms.

Let S be a finite set of atoms. A substitution θ is called a *unifier* for S if, for any pair of atoms A_1 and A_2 in S, applying the substitution θ yields $A_1\theta = A_2\theta$. A unifier θ for S is called a *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$. If θ is an mgu for A_1 and A_2, moreover, $A_1\theta = A_2$, then θ is a *term-matcher*.

Definition 2.2 Let a goal G be $\leftarrow A_1, \dots, A_m, \dots, A_k$ and a clause C be $A \leftarrow B_1, \dots, B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

- θ is an *mgu* of **the *selected*** atom A_m in G and A;
- G' is the goal $\leftarrow (A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k)\theta$.

A clause C_i^* is a *variant* of the clause C_i if $C_i^* = C_i\theta$, with θ being a variable renaming substitution such that variables in C_i^* do not appear in the derivation up to G_{i-1}. This process of renaming variables is called *standardising the variables apart*; we assume it throughout the paper without explicit mention.

Definition 2.3 An *SLD-derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \dots$ called *resolvents*, a sequence $C_1, C_2, \dots$ of variants of program clauses of P, and a sequence $\theta_1, \theta_2, \dots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ that has the empty clause $\Box$ as its last goal. If $G_n = \Box$, we say that the refutation has length n. The composition $\theta_1, \theta_2, \dots$ is called *computed answer*.

Depending on the algorithm behind the choice of the "selected atom", and behind the choice of the program clause for a resolvent, the proof-search strategy may differ. The most common strategy selects the left-most goal and top-most clause [19]. But the strategy may be changed to a random choice, cf. [17]. Also, there is an obvious choice between the breadth-first and depth-first search, if we view all the SLD-choices as a tree.

Example 2.4 An SLD-derivation for the goal `btree(X)`, with left-most atom, top-most clause and depth-first search is shown in the left side of Figure 1. Different strategies for that goal are represented in the right side of Figure 1.

If we pursue all possible selected atoms simultaneously instead of selecting one atom at a time, we will have an and-parallel implementation of the SLD-resolution. If we first pursue all possible clauses that unify with the given selected atom, we will have an or-parallel implementation. Pursuing both simultaneously gives and-or parallelism, see Figure 3 and [9, 10, 20].

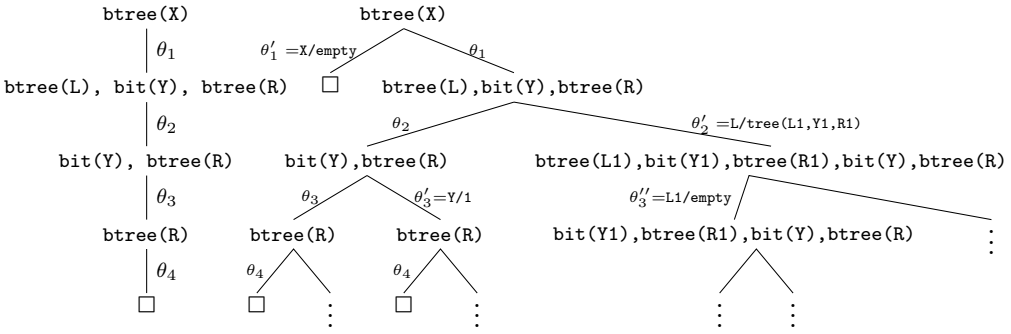

Fig. 1. **Left.** An SLD-derivation (also a refutation) for `BinaryTree` with goal `btree(X)`. The computed answer is given by the composition of θ_1 = `X/tree(L,Y,R)`, θ_2 = `L/empty`, θ_3 = `Y/0`, θ_4 = `R/empty`. **Right.** Different choices for SLD-derivations for the goal `btree(X)` selecting the left-most atom in a goal using a depth-first search strategy.

Example 2.5 [`BinaryTree`] The query `?- bit(X)` can be solved simultaneously with `bit(0)` and `bit(1)` using or-parallelism. For the query `?- btree(tree(L,X,R))`, an and-parallel algorithm can search for derivations for `btree(L)`, `bit(X)` and `btree(R)` simultaneously.

3 Parallel Derivations in Ground LP and Datalog

We first discuss parallel derivation strategies in the ground case. Consider the ground re-formulation of the program `BinaryTree`.

Example 3.1 [`BinaryTree - Ground Case`] This ground logic program (let us call it **BTG**) defines a subset of the set of binary trees presented in Example 2.1.

```
1. bit(0).
2. bit(1).
3. btree(empty).
4. btree(tree(empty,0,empty)) :- btree(empty), bit(0), btree(empty).
5. btree(tree(empty,1,empty)) :- btree(empty), bit(1), btree(empty).
```

3.1 Coalgebraic Semantics for Derivations in LP

Given a set At of propositions (atoms), [17] shows that there is a bijection between the set of variable-free logic programs over At and the set of $P_f P_f$-coalgebra structures on At, i.e., functions $p : At \longrightarrow P_f P_f(At)$, where P_f is the finite powerset functor: each atom of a logic program P is the head of finitely many clauses, and the body of each of those clauses contains finitely many atoms.

The endofunctor $P_f P_f$ necessarily has a cofree comonad $C(P_f P_f)$ on it. It has been noticed in [14], that, if a logic program can be modelled by a $P_f P_f$-coalgebra, then the SLD-derivations may be modelled by a comonad $C(P_f P_f)$ on this coalgebra. The main result of [14] established that, if $C(P_f P_f)$ is the cofree comonad on $P_f P_f$, then, given a ground (variable-free) logic program P, the induced $C(P_f P_f)$-coalgebra structure characterises the parallel and-or derivation trees (cf.

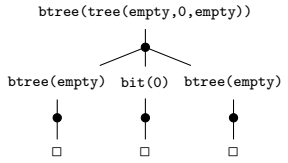

Fig. 2. Graphical presentation of the action of $\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$ on `btree(tree(empty,0,empty)`; this tree is also the and-or parallel derivation tree for `btree(tree(empty,0,empty)`.

[9]) of P. Here, we remind this construction, with a view of translating it first into a derivation algorithm, and then into implementation in Go.

Example 3.2 Consider the logic program **BTG** from Example 3.1. The program has five atoms, namely `bit(0)`, `bit(1)`, `btree(empty)`, `btree(tree(empty,0,empty))` and `btree(tree(empty,1,empty))`. So $At_{BTG} = \{$`bit(0)`, `bit(1)`, `btree(empty)`, `btree(tree(empty,0,empty))`, `btree(tree(empty,1,empty))`$\}$. And the program can be identified with the $P_f P_f$-coalgebra structure on At_{BTG} given by $p($`bit(0)`$) = \{\{\}\}$, $p($`bit(1)`$) = \{\{\}\}$, $p($`btree(empty)`$) = \{\{\}\}$ — where $\{\}$ is the empty set, and $\{\{\}\}$, is the one element set consisting of the empty set — $p($`btree(tree(empty,0,empty))` $= \{\{$`btree(empty), bit(0), btree(empty)`$\}\}$, $p($`btree(tree(empty,1,empty))` $= \{\{$`btree(empty), bit(1), btree(empty)`$\}\}$.

Let $C(P_f P_f)$ denote the cofree comonad on $P_f P_f$. For any set At, $C(P_f P_f)(\mathrm{At})$ is the limit of a diagram of the form

$$\ldots \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At} \times P_f P_f(\mathrm{At})) \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At}) \longrightarrow \mathrm{At}.$$

Given $p : \mathrm{At} \longrightarrow P_f P_f(\mathrm{At})$, put $\mathrm{At}_0 = \mathrm{At}$ and $\mathrm{At}_{n+1} = \mathrm{At} \times P_f P_f(\mathrm{At}_n)$, and consider the cone defined inductively as follows:

$$p_0 = id : \mathrm{At} \longrightarrow \mathrm{At}\ (= \mathrm{At}_0)$$
$$p_{n+1} = \langle id, P_f P_f(p_n) \circ p \rangle : \mathrm{At} \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At}_n)\ (= \mathrm{At}_{n+1})$$

The limiting property determines the coalgebra $\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$.

Example 3.3 Continuing the previous example, $\bar{p}($`btree(tree(empty,0,empty))`$) = p_2($`btree(tree(empty,0,empty))`$) = \langle$`btree(tree(empty,0,empty))` $\times \{\{\langle$`btree(empty)` $\times \{\{\}\}\rangle, \langle$`bit(0)` $\times \{\{\}\}\rangle, \langle$`btree(empty)` $\times \{\{\}\}\rangle\}\}\rangle$. This construction could be graphically represented as a tree, see Figure 2. If we think that every node of that tree is computed simultaneously and independently of the others, we may also say that Figure 2 shows the and-or parallel SLD-refutation for the goal `btree(tree(empty,0,empty))`.

In Figure 2, the nodes alternate between those labelled by atoms and those labelled by bullets (•). Bullets correspond to the number of sets contained in the outer set. In Example 3.3, the big outer set contains one set with three elements, hence the tree root in Figure 2 has one •-node child, followed by further three children nodes. We use the traditional notation □ to denote $\{\}$.

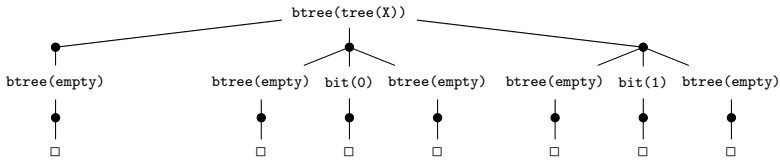

Fig. 3. The and-or parallel derivation tree for `btree(X)` for the `BTG` program.

3.2 From Semantics to Derivation Algorithm

The following definition, first formulated in [14], is CoALP's interpretation of and-or parallel derivations arising in Logic Programming, cf. [10].

Definition 3.4 Let P be a ground logic program and let $G = \leftarrow A$ be an atomic goal (possibly with variables). The *and-or parallel derivation tree* for A is the (possibly infinite) tree T satisfying the following properties.

- A is the root of T.
- Each node in T is either an and-node (an atom) or an or-node (given by $\bullet$).
- For every node A' occurring in T, if A' is unifiable with exactly $m > 0$ distinct clauses $C_1, \ldots, C_m$ in P (a clause C_i has the form $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, for some n_i) via mgu's $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in \{1, \ldots, m\}$, if $C_i = B^i \leftarrow B_1^i, \ldots, B_n^i$, then the ith or-node has n children given by and-nodes $B_1^i\theta_i, \ldots, B_n^i\theta_i$.

Example 3.5 An and-or tree corresponding to the **BTG** program with `btree(X)` as goal is shown in Figure 3.

The following "Constructive Completeness" result is a re-formulation of the more general soundness and completeness results of [14,17]. The below formulation serves our ultimate goal of tracing the inheritance of constructions from coalgebraic semantics to logic algorithm and data structures used in implementation.

Theorem 3.6 (Constructive Completeness) *Let P be a ground logic program, and G be a ground atomic goal. Given the construction of $\overline{p}(G)$, there exists (can be constructed) an and-or tree T_G for G, such that:*

- *(Tree depth 0.) The root of T_G is given by $p_0(G) = G$.*
- *(Tree depth n, for odd n.) Every node A appearing at the tree depth $n-1$ has m $\bullet$-child-nodes at the tree depth n, corresponding to the number of sets contained in the set $p(A)$.*
- *(Tree depth n, for even $n > 0$.) Every ith $\bullet$-node at the depth $n-1$ with a parent node A at the level $n-2$ has children at the depth n, given by the distinct elements of the ith set contained in the set $p(A)$.*

Moreover, T_G has finite depth $2n$ (for some $n \in \mathbb{N}$) iff $\bar{p}(G) = p_n(G)$. The T_G is infinite iff $\bar{p}(G)$ is given by the element of the limit $\lim_\omega(p_n)(At)$ of an infinite chain given by the construction of $C(P_f P_f)$ above.

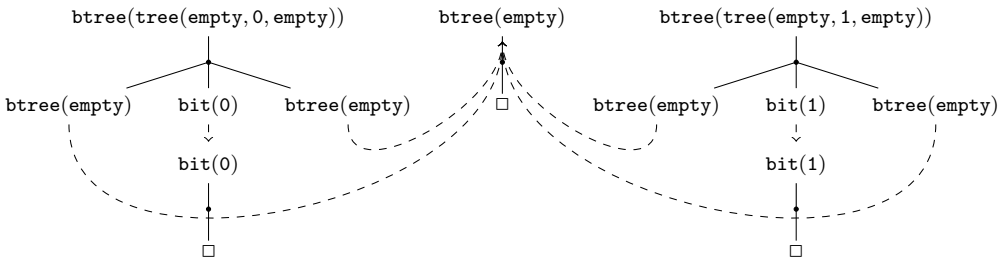

Fig. 4. Clause-trees for the BTG program with dashed lines denoting references between trees.

3.3 From Derivation Algorithm to Implementation

The above definition of the and-or-tree can give rise to an interpreter. CoALP's implementation in Go starts with the construction of template trees called *clause-trees* – they are generated from the input program and are the building blocks that will be used for the construction of and-or parallel derivation trees.

Definition 3.7 Let P be a ground logic program and let $C = A \leftarrow B_1, \ldots, B_n$ be a clause in P. The *clause-tree* for C is the tree T satisfying the following properties:

- A is the root of T.
- Each node in T is either an and-node (an atom) or an or-node (given by $\bullet$).
- A's child is given by an or-node. This or-node has n children given by and-nodes $B_1, \ldots, B_n$.
- For every clause C' occurring in P and for every and-node B_i, if B_i is unifiable with the head of C', then the node B_i contains a reference to the root of the clause-tree of C'.

Note that the first three items in the definition of the clause-tree mimic very closely the action of $P_f P_f$-coalgebra p on elements of At. Whereas the last item of the above definition paves the way for implementation of the $\bar{p}$ construction. The next example makes the connection clear.

Example 3.8 The clause-tree structures with open list references for the BTG program are shown in Figure 4. Compare with Examples 3.2 and 3.3.

CoALP's implementation parses and transforms each program clause into a clause-tree when the program is loaded. We use two kinds of structures to encode clause-trees: and-nodes and or-nodes. The or-node structure consists of a list of pointers to and-nodes. The and-node structure consists of a clause head, together with a list of or-nodes, and a list to track references to clause-tree root nodes called *open list*. Open lists play an important role in our implementation: they are used in a lazy fashion to add new or-nodes to the or-node list in the future. Or-nodes and and-nodes are linked by pointers and are allocated dynamically, see Figure 4.

The construction of clause-trees from a given program consists of two steps. In the first one, the clauses of the program are transformed into clause-trees with

an empty list of references in the and-nodes. After the transformation pass, each and-node corresponding to a clause body atom is visited again and its open list is populated with references to the unifiable clause-tree root nodes. This is a one time process at the initialisation and does not need to be done again for different queries.

Construction 3.1 (Derivations by Clause trees) *Given a program P and a goal atom $G = \leftarrow A$, a clause-tree derivation proceeds to construct the tree T, as follows:*

(i) *A root A for T is created as an and-node containing the goal atom.*

(ii) *The open list of the root A is constructed by adding references to all clause-trees that have the same root atom.*

(iii) *For each reference in an open list O of a node A' (where the corresponding atom equals the referenced root node's atom), a copy of the or-node below the referenced node and all its children in the clause-tree are added as a child to A'. The reference is then deleted from O.*

(iv) *This process continues until all references in all the open lists in the tree T have been processed.*

Example 3.9 Given the query `btree(tree(empty,0,empty))` in the **BTG** program, we construct the and-or parallel tree as follows. We start with a tree only consisting of the goal atom as root and-node. The reference to the clause-tree with root `btree(tree(empty,0,empty))` is added to this and-node open list as the respective atoms of the nodes are equal. Then, we start processing all nodes that have references to other clause-tree roots. We check whether the clause-tree root node that is referenced equals the currently processed nodes atom. If the equality has been verified, we can substitute the node in our tree with a copy of the referenced clause-tree. In this case after the initial root node is expanded this process is done for the two leave nodes `btree(empty)` and `bit(0)`. We continue this match and copy process until no nodes with references that match are left in our constructed tree. For our query `btree(tree(empty,0,empty))`, the resulting tree will look like the and-or parallel tree depicted in Figure 2, compare also with Example 3.3.

Lemma 3.10 *Let P be a ground logic program and G be a ground atomic goal. Then, the and-or parallel derivation tree for G is given by Construction 3.1.*

Constructive Completeness of Theorem 3.6 and the lemma above show the full chain starting from coalgebra and ending with implementation. It now remains to show that the resulting parallel language is indeed efficient.

Construction 3.1 permits parallelisation since no variable synchronization is required, and the order in which nodes of the tree are expanded is not relevant. Different expansion strategies ranging from sequential-depth and breadth-first up to fully parallel can be considered. This process in principle scales to the number of references to other clause-trees that the given tree has.

During the construction of the parallel and-or tree, it has to be checked whether it contains a subset of branches that constitute a proof for the query that is the

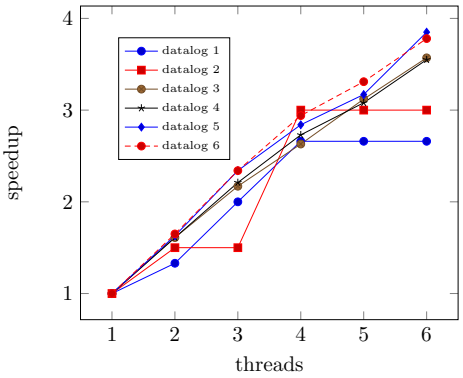

Fig. 5. Speedup of Datalog programs, relative to the base case with 1 thread, with different number of threads expanding the derivation tree.

root node of the tree. This work can be integrated into the expansion process and large parts can be done in parallel here as well.

Note that implementing the above restricted (ground) logic programs can have practical value of its own. Logic programs containing variables but no function symbols of arity $n > 0$ can all be soundly translated into finitely-presented ground logic programs. The most famous example of such a language is Datalog [12, 22]. The advantages of Datalog are easier implementations and a greater capacity for parallelisation. From the point of view of model theory, Datalog programs always have finite models.

Figure 5 shows the speedup that can be gained by constructing and-or parallel trees for Datalog programs in our system. The Datalog programs are randomly generated and can be examined in [18]. As can be seen in Figure 5, the speedup is significant and scales with the number of threads.

3.4 DataLog Case Study: BTG

Generally, given a query `btree(X)` and a ground variant of the `BinaryTree` program, we take the matching ground instances of `X` to construct its subtrees (cf. Figure 3). In the BTG fragment shown in Example 3.1, these are `btree(tree(empty,0,empty))`, `btree(tree(empty,1,empty))` and `btree(empty)`; but in some tests we describe here, there will be hundreds and even thousands of such instances. For each of these instances, parallel and-or trees can be constructed independently and in parallel. Our implementation provides several options to configure:

- *Number of threads.* This parameter indicates the number of threads that will be used in the general processing of the and-or parallel trees.
- *Parallel Expansion.* This option indicates that the expansion process will be run in parallel.
- *Number of expansion threads.* This is the number of total additional threads that will be used to help expand the and-or trees in parallel.

In order to test our parallel implementation and emulate the real-life database growth in Datalog, we have increased the number of clauses of the `BTG` program in two different ways. In Experiment 1, we use an algorithm (call it BTA) to generate hundreds of ground instances of clauses in `BinaryTree`, but making sure that the generated clauses describe balanced trees, i.e. having all the branches of the same depth. In the second experiment, we do not impose this restriction, and use an algorithm (call it UTA) that generates ground programs describing hundreds of (balanced and unbalanced) binary trees; we refer to the second kind of data as "unbalanced trees". The algorithms BTA and UTA are given in Appendix A.

There are various parameters to measure the success of a parallel language; we focus on three aspects, as follows.

1. Program speedup with the increase of parallel threads; or, in other words, *given a program, would its parallel execution bring significant speedup?* Figure 6 shows the speedup when increasing the number of threads and expansion threads, for ten BTG Datalog programs, of different sizes and nature. It also shows that the and-or derivation trees are generated faster when the *parallel expansion* option is activated. If the parallel expansion option is not used, increasing the number of threads does not significantly speed up the execution time (maximum speedup of 1.21); on the contrary, using the parallel expansion option and increasing the number of expansion threads considerably speeds up the execution time (maximum speedup of 4.13).

2. The gap between the best case and worst case of parallelisation: or *would any program be suitable for parallelisation?*

The BTA algorithm defines 2^{2^n-1} binary trees for a given depth n. Each of these trees will produce $2^{n+1}-1$ leaf nodes in the corresponding parallel and-or derivation tree, where n is the depth of the tree. The UTA algorithm generates $3 \cdot 6^{2 \cdot n-3} - 3 \cdot 6^{2 \cdot n-5}$ trees of depth n. E.g., for depth 3, UTA generates 630 binary trees, and BTA just 128. This means that the programs created with the UTA algorithm contain smaller trees (regarding number of nodes), and this impacts the speedup, as it is shown in the right diagram of Figure 6.

We notice several differences between the results obtained for programs generated with BTA and UTA algorithms. First of all, the number of clauses of the programs is bigger in the UTA cases and the number of leaves is similar in both BTA and UTA cases (cf. Figure 7); however, the runtime are considerable smaller for the UTA programs. This is due to the fact that the binary trees described by unbalanced Datalog programs are smaller (there are fewer leaves but more trees), hence, their parallel and-or trees are smaller and the derivation of the trees is faster.

Another difference is the impact of increasing the number of threads (or expand threads): for UTA programs, the maximum speedup is 1.78, which is not as good as for the BTA programs. The reason is again the size of the and-or trees. As the unbalanced trees are smaller, their creation is a small computational task and, therefore, the sequential overhead incurred by starting, syncing and distributing work among threads cannot be offset by working in parallel.

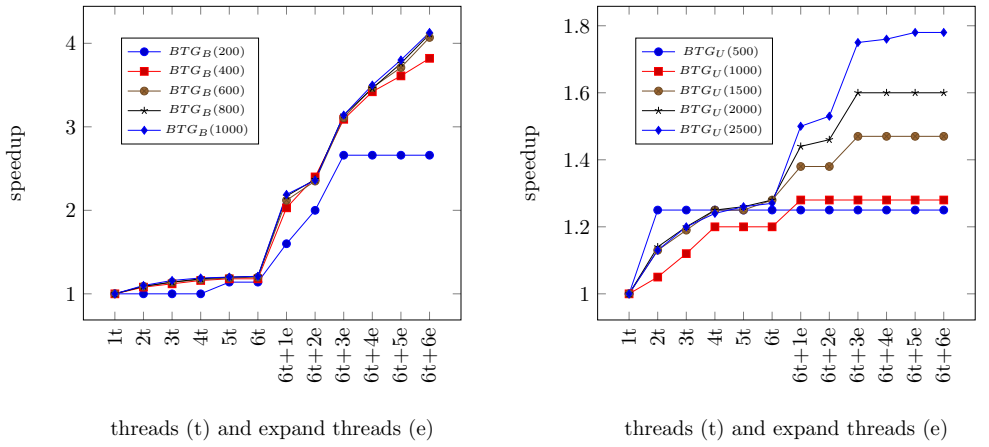

Fig. 6. Speedup for ten different Datalog versions of the `BTG` program, with different parameters. **Left.** Speedup of programs generated with the `BTA` algorithm. **Right.** Speedup of programs generated with the `UTA` algorithm. The values X of $\texttt{BTG}_B(X)$ and $\texttt{BTG}_U(X)$ indicate the number of clauses in the Datalog program.

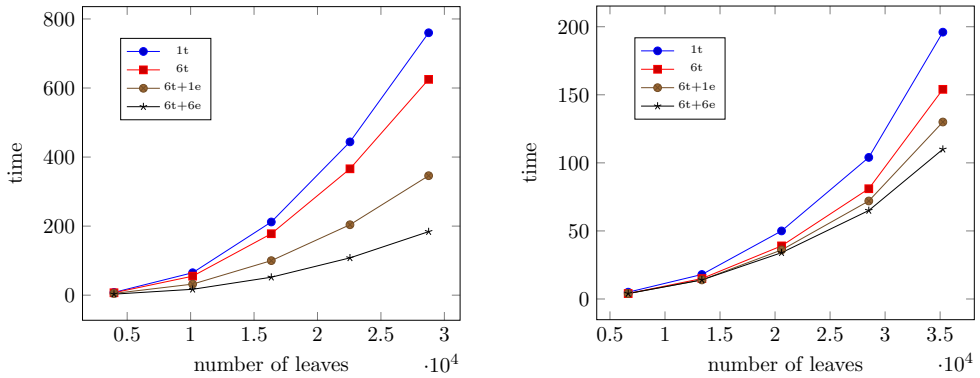

Fig. 7. Effect of parallelism on growing Datalog programs. **Left.** Time (in seconds) for the first benchmark (balanced trees). **Right.** Time (in seconds) for the second benchmark (unbalanced trees).

Apart from the size, there is another reason which prevents the speedup when increasing the number of expand threads for the UTA programs. In principle, keeping every expansion thread busy expanding a concrete part of a tree instead of directing it to work in different parts of the tree results in the best speedup. An ideal implementation would adapt to the tree shape and size, and would dedicate new expansion threads only for computations of sufficiently large parts of the tree. However, in unbalanced trees, it cannot be known in advance if part of a tree is large enough to offset the setup costs of dedicating a new thread to it, instead of just executing the work in the current thread. As a result, when a new thread is dedicated to expand a part of a tree that is not big enough, the expansion process is slowed down and execution time increases.

3. Effect of data growth or *would parallelisation bring benefits when the database described by the program increases?* Suppose one works with dynamically growing data and does not know in advance whether the database will eventually

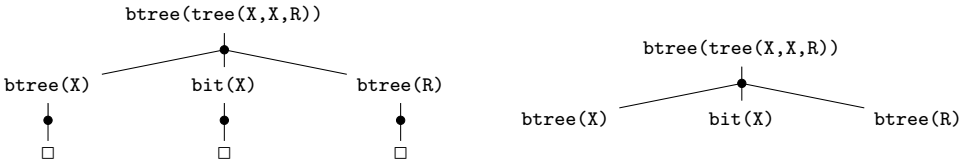

Fig. 8. **Left:** The naive first-order parallelisation leading to unsound substitutions, if two branches of the tree are allowed to substitute for X independently and in parallel. **Right**: the action of $\bar{p}$ on the goal `btree(tree(X,X,R))`; also, a coinductive tree for `btree(tree(X,X,R))`.

resemble a well-parallelisable case like BTA or a badly parallelisable case like UTA. Would CoALP do any good in the pessimistic scenario? Figure 6 shows that the speedup improves with the growth of the database: UTA programs of size 500 and 1000 clauses hardly allow any speedup, but with 2500 clauses, the speedup is nearing 2 times. Figure 7 studies this effect under a different angle: assuming increase in database and hence the number of and-or tree leaves, would parallel execution reduce the growth of execution time? – and it does for both BTA and UTA experiments.

4 Fibrational Semantics for Parallelism

We proceed to extend our coalgebraic approach to the general first-order case. Unification and SLD-resolution algorithms are P-complete in the general case [7,12]. In practical terms, P-completeness of an algorithm means that its parallel implementation would not provide effective speedup. The problem can be illustrated using the following example.

Example 4.1 The sequential derivation for the goal `?- btree(tree(X,X,R))` fails due to ill-typing. But, if the proof search proceeds in parallel fashion, it may find substitutions for `X` in distinct parallel and-branches of the derivation tree, see Figure 8. These substitutions will give an unsound result.

Therefore, implementations of *parallel* SLD-derivations require keeping special records of previously made substitutions and, hence, involve additional data structures and algorithms that coordinate variable substitution in different branches of parallel derivation trees [10]. This reduces LP capacity for parallelisation.

The practical response to this problem (cf. [10]), has been to distinguish cases where parallel SLD-derivations can be sound without synchronisation, and achieve efficient parallelisation there. There are two kinds of and-parallelism: *independent* and-parallelism and *dependent* and-parallelism. The former arises when, given two or more subgoals, there is no common variable in these goals (cf. Example 2.1). On the contrary, *dependent* and-parallelism appears when two or more subgoals have a common variable and compete in the creation of bindings for such a variable (cf. Example 4.1). Dependent and-parallelism can produce unsound derivations due to variable dependencies; hence, and-parallel implementations have to sacrifice parallelism for soundness and synchronise dependent variables. A different approach is pursued by CoALP [15,16] below.

4.1 Coalgebraic Semantics for First-order Parallel Derivations

We briefly recall the constructions involved in the coalgebraic semantics for first-order logic programs. We then translate them into derivation algorithms and implementation, following the same scheme as in the previous section.

Following the standard practice, we model the first-order language underlying a logic program by a Lawvere theory [1,3,5].

Definition 4.2 Given a signature Σ of function symbols, the *Lawvere theory* $\mathcal{L}_\Sigma$ generated by Σ is the following category: $\mathtt{ob}(\mathcal{L}_\Sigma)$ is the set of natural numbers. For each natural number n, let $x_1, \ldots, x_n$ be a specified list of distinct variables. Define $\mathtt{ob}(\mathcal{L}_\Sigma)(n, m)$ to be the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$. Define composition in $\mathcal{L}_\Sigma$ by substitution.

Example 4.3 Consider `BinaryTree`. The constants `0`, `1` and `empty` are modelled by maps from 0 to 1 in $\mathcal{L}_\Sigma$, and `tree` is modelled by a map from 3 to 1. The term `tree(0,0,empty)` is therefore modelled by the map from 0 to 1 given by the composite of the maps modelling `tree`, `0` and `empty`.

For each signature Σ, we extend the set At of atoms for a ground logic program to the functor $At : \mathcal{L}_\Sigma^{op} \to Set$ that sends a natural number n to the set of all atomic formulae generated by Σ, variables among a fixed set $x_1, \ldots, x_n$, and the predicate symbols appearing in the logic program. A map $f : n \to m$ in $\mathcal{L}_\Sigma$ is sent to the function $At(f) : At(m) \to At(n)$ that sends an atomic formula $A(x_1, \ldots, x_m)$ to $A(f_1(x_1, \ldots, x_n)/x_1, \ldots, f_m(x_1, \ldots, x_n)/x_m)$, i.e., $At(f)$ is defined by substitution.

Example 4.4 For `BinaryTree`, $At(3)$ is a poset containing: `bit(0)`, `bit(1)`, `btree(empty)`, `btree(tree(L,X,R))`, `btree(L)`, `bit(X)`, `btree(tree(0,0,0))`, `btree(tree(1,1,1))`, `btree(tree(0,empty,0))`, ..., `bit(tree(0,0,0))`, `bit(tree(1,1,1))`, `bit(tree(0,empty,0))`, ..., `btree(tree(tree(L,X,R)),X,tree(L,X,R))`, `btree(tree(tree(X,X,X)),X,tree(X,X,X))`, ... Due to the presence of the function symbol `tree` that can be composed recursively, it is an infinite set. Notice the restriction on the number of distinct variables in the fibre of 3.

Given a logic program P with function symbols in Σ, [16, 17] model P by the $Lax(\mathcal{L}_\Sigma^{op}, Poset)$-coalgebra, whose n-component takes an atomic formula $A(x_1, \ldots, x_n)$ with at most n variables, considers all substitutions of clauses in P whose head *agrees* with $A(x_1, \ldots, x_n)$, and gives the set of sets of atomic formulae in antecedents. We say a head H (from a clause $H \leftarrow body$) agrees with $A(x_1, \ldots, x_n)$ if the following conditions hold:

(i) $H\theta = A(x_1, ..., x_n)$

(ii) applying θ to *body* yields formulae all of whose variables must be among $x_1, \ldots, x_n$.

The conditions (i, ii) above deserve careful discussion, as they have implications on implementation of CoALP. They are necessary and sufficient conditions to in-

sure that the fibrational discipline is obeyed in the coalgebraic model. Note that condition (i) resembles very much the definition of term-matching (cf. Section 2). The asymmetric application of the substitution θ is not the only feature that distinguishes the above items from most general unifiers (mgus): item (i) does not require θ to be "most general", which also distinguishes (i) from term-matching. Item (ii) insures no new variables have been introduced within one fibre.

Example 4.5 Continuing Example 4.4 for the program `BinaryTree`, for $At(0)$, the $Lax(\mathcal{L}_\Sigma^{op}, Poset)$-coalgebra will essentially give account to the version of `BTG` program in which all possible ground instances of `BinaryTree` are present in the form of clauses: e.g, for `btree(tree(empty,0,empty))` $\in At(0)$, $p(0)$(`btree(tree(empty,0,empty))`) = {{`btree(empty), bit(0), btree(empty)`}}. For `btree(tree(tree(empty,0,empty),0,tree(empty,0,empty)))` $\in At(0)$, $p(0)$(`btree(tree(tree(empty,0,empty),0,tree(empty,0,empty)))`) = {{`btree(tree(empty,0,empty)), bit(0), btree(tree(empty,0,empty))`}}.

But note that the empty set will correspond to e.g. $p(1)$(`bit(X)`), as no clause in `BinaryTree` *agrees* with it; cf. Item (i).

The next example shows the effect of items $(i) - (ii)$ on the derivations.

Example 4.6 Consider the following program (we call it TQ):

```
1. T(X,c) :- Q(X).
2. Q(X) :- P(X).
3. Q(a).
4. P(b) :- P(X).
```

For $Q(a) \in At(0)$, $p(0)(Q(a)) = \{\{P(a)\}, \{\}\}$, the last set $\{\}$ models the third clause. However, for $Q(X) \in At(1)$, the corresponding set will be $p(1)(Q(X)) = \{\{P(X)\}\}$. Item (ii) plays a role with clauses introducing a new variable in the body, like clause 4. For fiber of 0, $p(0)(P(b)) = \{\{P(a)\}, \{P(b)\}, \{P(c)\}\}$, note that both the fact that θ is not required to be an mgu, and the item (ii) play a role here. See also Figure 9, the right-hand side.

Similarly to the ground case, this can be extended to model derivations. A lax natural transformation $\bar{p} : At \longrightarrow C(P_c P_f)At$, when evaluated at n, is the function from the set $At(n)$ to the set of all possible derivations in a fibre n. Note the P_c – a countable powerset functor – is introduced to model countable signature, as will be further explained below. The role of laxness is explained in detail in [16,17] and critiqued in [4], so we will not focus on this issue here. Instead, we will concentrate on the role of the *fibrational semantics* in the design of parallel derivation algorithm.

Example 4.7 Consider `BinaryTree` as in Example 2.1. Suppose we start with an atomic formula `btree(tree(X,X,R))` $\in At(2)$. Then $\bar{p}$(`btree(tree(X,X,R))`) is the element of $C(P_c P_f)At(2)$ expressible by the tree on the right hand side of Figure 8. Note that despite there being an infinite number of binary trees and the infinite number of instances of clauses of `Binary Tree`, the countability accounted

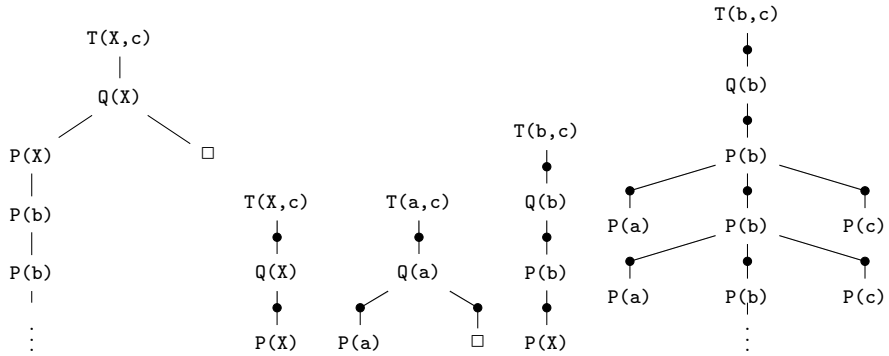

Fig. 9. **Far Left**. The SLD-tree for TQ and the goal `T(X,c)`. **Centre.** The coinductive trees for TQ and the goals `T(X,c)`, `T(a,c)`, `T(b,c)`. For `T(X,c)` and `T(a,c)` the coinductive trees correspond to the action of $\bar{p}(1)(T(X,c))$ and $\bar{p}(0)(T(a,c))$. **Far Right.** Action of $\bar{p}(0)$(`T(b,c)`).

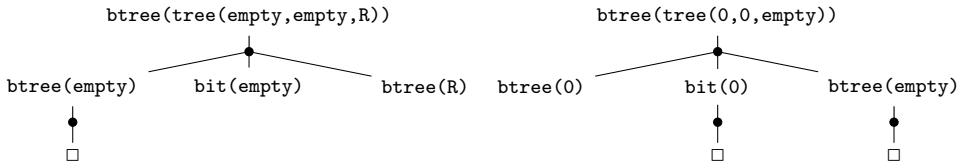

Fig. 10. Action of $\bar{p}(1)$ on `btree(tree(empty,empty,R))` and `btree(tree(0,0,empty))`; also, coinductive trees for `btree(tree(empty,empty,R))` and `btree(tree(0,0,empty))`.

for by P_c does not arise for programs of this kind, cf. Items $(i-ii)$.

This tree agrees partially with the and-or parallel derivation tree for `btree(tree(X,X,R))` given on the left of Figure 8. But it has leaves `btree(X)`, `bit(X)` and `btree(R)` (cf. Item (i)), whereas the and-or parallel derivation tree follows those nodes, using substitutions determined by mgu's that might not be consistent with each other, e.g., there is no consistent substitution for `X`.

Action of $\bar{p}(1)$(`btree(tree(empty,empty,R))`) and $\bar{p}(1)$(`btree(tree(0,0,empty))`), are shown in Figure 10. Compare with Examples 4.4 and 4.5; again, note the effect of Item (i).

Example 4.8 The action of $\bar{p}(0)T(b,c)$ is given in Figure 9. Note the infinite depth. If there was a constructor in the language, e.g. `f`, then the branching could be infinite, bringing `P(f(a))`, `P(f(f(a))`, ..., `P(f(b))`, ... in the fibre of 0, alongside the branches with `P(a)`, `P(b)`, and `P(c)`. This is the effect of Item (i) not requiring θ to be an mgu. Programs of this kind require countability in P_cP_f.

We define a clause to be *regular*, if the set of variables appearing in its body is a proper subset of the variables appearing in its head. We define logic program to be a *regular program*, if all of its clauses are regular; otherwise the program is *irregular*.

4.2 From Semantics to Derivation Algorithm

The above coalgebraic semantics suggests to restrict *unification* involved in e.g. SLD-derivations to *term matching*. However, for constructive reasons, we do not follow the above coalgebraic semantics literally this time, and do not lift restriction of the term-matcher to be the mgu. This permits to avoid infinite cycles when working with irregular programs, cf. Figure 9. To compensate, we relax the restriction on body variables (cf. Item (*ii*)) and allow new variables to be introduced within the derivations.

Definition 4.9 Let P be a logic program and $G = \leftarrow A$ be an atomic goal. The *coinductive tree* for A is a possibly infinite tree T satisfying the following properties.

- A is the root of T.
- Each node in T is either an and-node (an atom) or an or-node (given by $\bullet$).
- For every and-node A' occurring in T, if there exist exactly $m > 0$ distinct clauses $C_1, \ldots, C_m$ in P (a clause C_i has the form $B_i \leftarrow B_1^i, \ldots, B_{n_i}^i$, for some n_i), such that $A' = B_1\theta_1 = ... = B_m\theta_m$, for mgus $\theta_1, \ldots, \theta_m$, then A' has exactly m children given by or-nodes, such that, for every $i \in m$, the ith or-node has n_i children given by and-nodes $B_1^i\theta_i, \ldots, B_{n_i}^i\theta_i$.

Example 4.10 Figures 8–10 (excluding Figure 9) showing the action of $\bar{p}$ on program atoms, would correspond exactly to coinductive trees for these atoms. Note the difference between the coinductive trees and and-or parallel trees (Figure 8) and the SLD-derivations for `BinaryTree` (Figure 1). However, there will be cases when the coinductive trees and the corresponding action of $\bar{p}$ differ, cf. Figure 9.

The seemingly small restriction of unification to term-matching changes the way the proof-search is handled within each coinductive tree. Note that unification in general is inherently sequential, whereas term matching is parallelisable [7]. Term-matching permits implicit handling of both parallelism and corecursion:

- due to term-matching, all existing variables are not instantiated and will remain the same within one tree, and therefore no explicit variable synchronisation is needed when (the branches of) trees are expanded in parallel.
- term-matching permits to unfold coinductive trees lazily, keeping each individual tree at a finite size, provided the program is well-founded [15,17]. Laziness in its turn plays a role in delaying substitutions in parallel derivations.

Example 4.11 Consider the program TQ from Example 4.6. Figure 9 shows the SLD-derivations and the coinductive tree for goal $T(X, c)$. The SLD-derivations produce one derivation that loops forever; and, subject to clause re-ordering or loop-termination, would also give one answer computed on the second step of derivations. The coinductive tree for the same goal stops lazily, and neither loops nor gives the answer. However, after a suitable substitution, the coinductive tree for `T(a,c)` gives the sought answer (we will use this technique for coinductive derivations in the next section). The coinductive trees for the goal atoms `T(X,c)` and `T(a,c)` in Figure 9 correspond to the action of $\bar{p}$ on them. However, the case of `T(b,c)` is different.

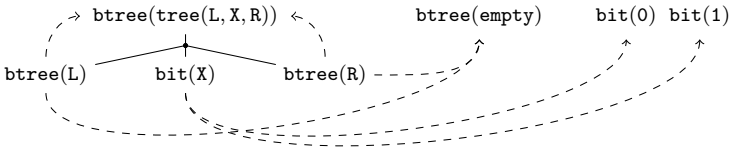

Fig. 11. Clause-trees for the `BinaryTree` program with dashed lines denoting open list references.

Note the loop stopping lazily in the coinductive tree for `T(b,c)`, which distinguishes it from infinite SLD-derivations and the construction of $\bar{p}(0)(\texttt{T(b,c)})$.

In line with Section 3, we would like to establish a *constructive completeness result*, showing how to transform our coalgebraic semantics into coinductive trees. However, this is impossible; Examples 4.6 and 4.11, and Figure 9 show a counter-example. In fact, in [15,17], completeness theorem for CoALP gives a weak, rather than a strong (constructive), completeness statement. We can, however, establish constructive completeness for regular LPs.

Lemma 4.12 (Restricted Constructive Completeness) *Let P be a regular logic program, and G be an atomic goal with exactly s variables. Given the construction of $\overline{p}(s)(G)$, there exists (can be constructed) a coinductive tree T_G for G, such that:*

- *(Tree depth 0.) The root of T_G is given by $p_0(s)(G) = G$.*
- *(Tree depth n, for odd n.) Every node A appearing at the tree depth $n-1$ has m $\bullet$-child-nodes at the tree depth n, corresponding to the number of sets contained in the set $p(s)(A)$.*
- *(Tree depth n, for even $n > 0$.) Every ith $\bullet$-node at the depth $n-1$ with a parent node A at the depth $n-2$ has children at the depth n, given by the distinct elements of the ith set in $p(s)(A)$.*

Moreover, T_G has finite depth $2n$ (for some $n \in \mathcal{N}$) iff $\bar{p}(G) = p_n(s)(G)$. The T_G is infinite iff $\bar{p}(s)(G)$ is given by the element of the limit $\lim_\omega(p_n)(s)(At)$ of an infinite chain given by Construction of $C(P_f P_f)$.

Proof. The core of this inductive proof is an observation that, if θ is restricted to the variables of A as it effectively happens for regular programs, and $A\theta = B$, then necessarily θ is an mgu. This fact eliminates the difference between Item (i) and the term-matching for coinductive trees. Further, Item (ii) is redundant for regular programs. Note that we use $P_f P_f$, rather than $P_c P_f$ here. □

4.3 From Derivation Algorithm to Implementation

The CoALP implementation of coinductive trees reuses the notion of clause-trees (cf. Definition 3.7) presented in Subsection 3.3.

Example 4.13 Figure 11 shows the clause-trees for all the clauses of the `BinaryTree` program.

Clause-trees of Definition 3.7 remain to be the building blocks of coinductive

trees. However, the generation of coinductive trees from clause-trees is slightly different to the one presented in Construction 3.1. We assume all programs are pre-processed this way.

Construction 4.1 (Go-coinductive tree) *Given a logic program P and a goal $G = \leftarrow A$, generate a Go-coinductive tree T as follows:*

(i) *A root A for T is created as an and-node containing the goal atom.*

(ii) *The open list of the root A is constructed by adding references to all clause-trees that have a unifiable root atom.*

(iii) *For each reference in an open list O of a node A' where the corresponding atom matches the referenced root node's atom R, a copy of the or-node below the referenced node and all its children in the clause-tree are added as child to A'. All the substitutions that were needed to make R match are also applied to the newly copied node atoms. The reference is then deleted from O.*

(iv) *This process continues until all references in all the open lists in the tree T have been processed.*

The first difference to the construction of and-or parallel trees is that instead of needing equality to the root node of a referenced tree its now only required to be term matching in order to expand the tree. The second difference is that the newly added nodes to the tree require application of the substitutions needed for term-matching of the clause-tree they belong to. Since requiring equality in the Construction 3.1 did not generate substitutions, this step was previously unnecessary.

Example 4.14 Given the query `btree(tree(X,X,R))` in the `BinaryTree` program, we construct its Go-coinductive tree as follows. We start with a tree that consists of the goal atom as root and-node and no or-node children. Then, we add references to all clause-trees with unifiable root node atom — in this case is the clause-tree for `btree(tree(L,X,R))`. For this root to match the query, the substitution `L/X` is needed. Therefore, we copy the or-node and its and-node children from the clause-tree for `btree(tree(L,X,R))` below our tree root and apply the aforementioned substitution to the copied clause-tree nodes. Now, we should process all nodes in this newly created tree that have references to other clause-tree roots, but there are no nodes with references that match. Therefore, the process is finished, and the resulting tree will look like the one depicted in the right side of Figure 8.

Lemma 4.15 *Let P be a logic program and let G be an atomic goal. Then, the coinductive tree of G is given by Construction 4.1.*

We employ an optimisation technique to minimise the work for constructing Go-coinductive trees. When checking a referenced root node for term-matching, it is also checked whether it can be unified. If it is not unifiable, then term-matching is impossible even if substitutions will be applied later when processing the tree. Therefore, non-unifiable references to clause-tree roots can be immediately removed from the open lists. This process of filtering open lists and copying or-nodes and their child nodes during tree expansion can be done in parallel as no variable sub-

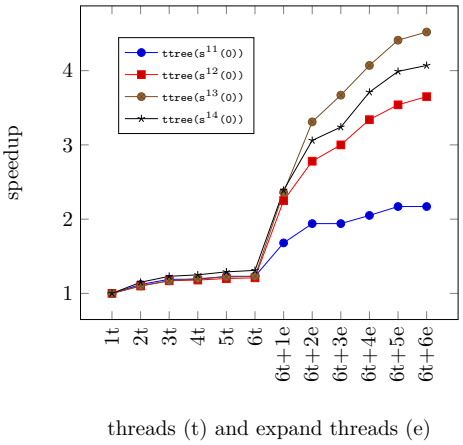

Fig. 12. Speedup of the construction of the coinductive tree of `ttree(s^i(0))`, relative to the base case with 1 thread, for different values of i and different parameters.

stitutions in existing atoms is required. It only needs to be guaranteed that only one thread changes the properties of one node concurrently. In the same way, no run-time coordination for tree merging or variable substitution is needed. However, the parallel expansion option should only be employed on large trees to amortise the additional overhead of dispatching and managing multiple threads that execute the mentioned tasks.

4.4 Case Study: Ttree

The construction of coinductive trees for queries in the `BinaryTree` program is too fast to notice any benefit from parallelism; instead we introduce a new example.

Example 4.16 [`Ttree`] This program has three and-parallel branches and thereby can be used to construct coinductive trees with three branches at each non leaf or-node.

```
1. ttree(0).
2. ttree(s(X)) :- ttree(X), ttree(X), ttree(X).
```

We benchmark our implementation by constructing the tree and thereby proof for the query `ttree(s^i(0))`, where i indicates the number of times that the function `s` is nested (e.g. `ttree(s^2(0))` is equal to `ttree(s(s(0)))`). Given the query `ttree(s^i(0))`, the associated coinductive tree will have 3^i leaf nodes. Therefore, we can expect that the construction of coinductive trees will get advantage of a parallel expansion; this is confirmed in Figure 12.

The construction of coinductive trees clearly speeds up with introduction of parallelism. Increasing the number of threads speeds up the execution time whether the parallel expand option is activated (maximum speedup of 4.52), or not (maximum speedup of 1.31). Note that generalising from ground logic programs to first-order logic programs did not reduce the best-case scenario speedup of ≈ 4.5.

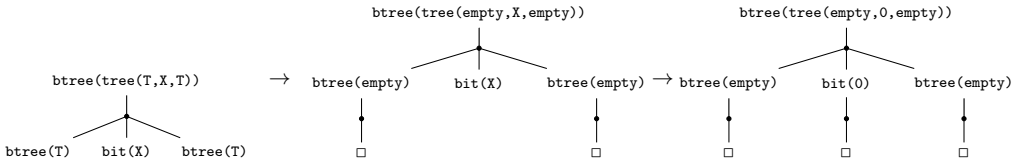

Fig. 13. A coinductive derivation for the goal `btree(tree(T,X,T))` and the program `BinaryTree`. Note that one tree cannot find the correct answer (substitution), but it needs two steps to compute one possible answer. The first transition is given by $\theta_0 = T/\texttt{empty}$, the second by $\theta_1 = X/0$.

5 Derivations by Coinductive Trees

In the previous sections, we have seen how first-order atoms, clauses, as well as individual derivations can be modeled using category theoretic constructs. We have taken inspiration from the coalgebraic semantics to introduce the notion of coinductive trees. However, as can be seen from e.g. Figures 9 and 10, one coinductive tree may not produce the answer corresponding to a refutation by the SLD-resolution. Instead, a sequence of coinductive trees may be needed to advance the derivation. In this section, we introduce the derivations involving coinductive trees, and discuss their relation to the coalgebraic semantics. The coalgebraic derivation algorithm follows closely [15,17], but the implementation we present here is new.

5.1 Coinductive Derivations: relating semantics to derivations

We start with composing coinductive trees into derivations.

Definition 5.1 Let G be a goal given by an atom $\leftarrow A$ and the coinductive tree T induced by A, and let C be a clause $H \leftarrow B_1, \ldots, B_n$. Then goal G' is *coinductively derived* from G and C using mgu θ if the following conditions hold:

- A' is an atom in T.
- θ is an *mgu* of A' and H.
- G' is given by the atom $\leftarrow A\theta$ and the coinductive tree T' determined by $A\theta$.

Coinductive derivations resemble *tree rewriting*, an example is shown in Figure 13. They produce the "lazy" corecursive effect: derivations are given by potentially infinite number of steps, where each individual step (coinductive tree) is executed in finite time. The ultimate goal of derivations is to find success (sub)trees.

Definition 5.2 Let P be a logic program, G be an atomic goal, and T be a coinductive tree determined by P and G. A subtree T' of T is called a *coinductive subtree* of T if it satisfies the following conditions:

- the root of T' is the root of T (up to variable renaming);
- if an and-node belongs to T', then one of its children belongs to T'.
- if an or-node belongs to T', then all its children belong to T'.

A finite coinductive (sub)tree is called a *success (sub)tree* if its leaves are empty goals (equivalently, they are followed only by $\Box$ in the usual pictures).

Definition of coinductive derivations and refutations is an adaptation of Definition 2.3, modulo Definitions 5.1 and 5.2.

Example 5.3 Figure 13 shows an example of a coinductive derivation. The last coinductive tree is also a success subtree in itself; which means the derivation has been successful.

Transitions between coinductive trees can be done in a sequential or parallel manner, as we discuss in the next section. In [17], we have proven the soundness and completeness of the coalgebraic derivations relative to the coalgebraic semantics of [14, 16]. What would constructive completeness result say here? It would have to build upon Lemma 4.12 when it comes to modelling individual trees in the derivations. In addition, we would need to produce a construction (algorithm) that, for any goal $G(x_1, \dots x_k)$ and any map θ in $\mathcal{L}_\Sigma^{op}$ (corresponding to a substitution) would produce a coinductive derivation starting at $G(x_1, \dots, x_k)$ and ending at $G(x_1, \dots, x_k)\theta$, θ being a computed substitution. However, this result would be analogous to proving decidability of entailment for the Horn-clause LP, whereas it is only semi-decidable. This is why, completeness is not generally stated in a constructive form for LP and instead involves the nonconstructive existence assertion, see [19] for standard completeness statements and [17] for CoALP.

5.2 From Derivation Algorithm to Implementation

The construction of coinductive derivations is modelled as a search through the graph of coinductive trees connected by the derivation operation. To keep track of which trees have to be processed, the implementation maintains an ordered list of coinductive trees that is called the *work queue*. Initially, this list will be filled with the coinductive tree constructed from the input goal. The top level control-flow-loop dispatches coinductive trees from the work queue to be processed simultaneously by multiple worker threads. Worker threads are implemented by goroutines that are lightweight threads of execution and communicate via Go's channels for passing values. Each worker executes the following simplified steps independently on the received tree:

- checks and reports if the tree contains a success subtree,
- finds nodes with a non-empty open list and compute the set of distinct *mgu's* needed to unify with the referenced clause-tree roots,
- for each *mgu* found, applies the *mgu* to a copy of the tree then expands it by the process outlined in construction 4.1,
- sends the created trees back to the main control loop to be added back to the work pool for further processing.

This process can be run with a variable number of worker threads until either the requested amount of success subtrees are found or the work queue is empty and all the possible coinductive trees for the query have been processed. This search process requires to take some decisions about strategies to manage the work queue,

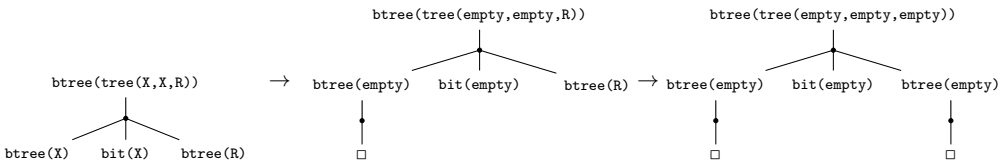

Fig. 14. A coinductive derivation for the goal `btree(tree(X,X,R))`. Parallel expansion of nodes in a coinductive tree does not lead to unsound substitutions.

find open nodes, compacting coinductive trees and ordering solutions.

Organisation of the work queue. The work queue is sufficient to keep track of all trees which have derivation steps that still need to be evaluated. This list of coinductive trees can be managed either as a first-in-first-out queue or alternatively as a last-in-first-out stack. This determines the search strategy that is employed to find success trees in the possible derivation chains: depth-first in the case of last-in-first-out stack (this is the strategy followed in PROLOG), and breadth-first in the case of first-in-first-out queue (which is the strategy followed in CoALP).

A depth-first search strategy has the drawback of not finding some possible success trees if the search forever follows an unrelated infinite derivation chain. By construction, coinductive derivation trees are always finite; however, this does not restrict the possibility of an infinite chain of derivations between trees. The breadth-first search is the strategy currently chosen in CoALP and therefore no traditional backtracking is employed in contrast to PROLOG. This has the usual drawback of requiring more memory than a depth-first search. However, it allows CoALP to easily report solutions sorted by the number of substitutions. If a solution with a specific amount of substitutions is reported, it can be guaranteed that no solution with a lesser amount of substitutions will be found later.

Strategies to find open nodes. Open nodes are and-nodes which contain at least one reference in their open list which points to a root node of a clause-tree with a unifiable atom. Not only leave nodes in the tree need to be checked but any node. An example program which can give rise to such a situation can be seen in Example 4.6 where the node for $\mathtt{Q(X)}$ in the tree for the goal $\mathtt{T(X,c)}$ contains a non empty open list with a reference to $\mathtt{Q(1)}$. There are different strategies to find open nodes. If the left- or right-most deepest branch is always searched for open nodes, this would mimic PROLOG's depth-first search approach to SLD-resolution. To obtain solutions not found in the depth-first manner, a breadth-first search approach to find open nodes is employed. It always chooses the left most node on the lowest level possible. This will yield a transition to each possible reachable coinductive tree after a finite number of derivations. This is especially important when dealing with co-inductive or cyclic data structures. If no open node in the coinductive tree can be found, the tree will simply be discarded as no further derivations are possible.

Example 5.4 In Figure 14, the coinductive derivation tree for the goal `btree(tree(X,X,R))` is shown on the left. The open leaf on the lowest level to the left is `btree(X)` with the tree templates with the goals `tree(empty)` and $\mathtt{tree(L_1,B_1,R_1)}$. Therefore the distinct *mgu's* are {`X/empty`}

and $\{\texttt{X/tree}(\texttt{L}_1, \texttt{B}_1, \texttt{R}_1)\}$. They yield different derivation trees after expansion. The one for $\{\texttt{X/empty}\}$ is depicted in the middle of Figure 14. Note that for this tree no expansion was needed as no nodes with open lists have new term matching goals.

Compacting and pruning coinductive trees. To minimise the amount of used memory and to avoid unnecessary copying of nodes, the implementation contains various mechanisms to remove nodes from the trees in the internal data structures – these nodes are guaranteed to be irrelevant for further derivations and to determine whether the tree contains a success subtree. For each found *mgu* for the selected open node, a distinct copy of the coinductive tree is created and the unifier applied, afterwards this new tree is expanded. Multiple generated coinductive trees represent different branches in the coinductive derivation process. If only one branch is needed, the original tree is reused to avoid an unnecessary copy, and the substitution is applied directly to it – this speeds up non branching clauses. During the copy process, trees are pruned by removing success and non-succeeding subtrees. Also, chains of single or-nodes are shortened by removing intermediate and-nodes with empty open leaves. These optimisations correspond to the trimming of stack frames (e.g. tail call optimisation) – a technique employed in PROLOG.

Ordering of solutions. Due to the nature of pre-emptive threading, CPU cores working at different speeds, memory access having varying latencies, and trees requiring different computational effort; the order of returned trees in the work queue is not deterministic and, therefore, it will change if more than one worker thread is involved. Using the substitution length of all the substitutions in the derivation chain as priority ranking, we gain an enumeration order even for a potentially infinite lazy derivation processes and the implementation can report solutions in this order.

Example 5.5 While an infinite number of coinductive trees can in principle be produced for the goal `btree(X)`, the algorithm returns the solutions for the first five success trees in finite time in the following order:
`btree(empty)` with `X/empty`
`btree(tree(empty, 0, empty))` with `X/tree(L1,B1,R1)`, `L1/empty`, `B1/0`, `R1/empty`.
`btree(tree(empty, 1, empty))` with `X/tree(L1,B1,R1)`, `L1/empty`, `B1/1`, `R1/empty`.
`btree(tree(tree(empty, 0, empty), 0, empty))` with `X/tree(L1,B1,R1)`, `L1/tree(L2,B2,R2)`, `L2/empty`, `B2/0`, `R2/empty`, `B1/0`, `R1/empty`.
`btree(tree(empty, 0, tree(empty, 0, empty)))` with `X/tree(L1,B1,R1)`, `L1/empty`, `B1/0`, `R1/tree(L2,B2,R2)`, `L2/empty`, `B2/0`, `R2/empty`.

This is implemented by buffering success trees in a priority queue. If it is determined that the work queue and all workers only hold derivation trees with the same or higher number of derivation steps, the solutions up to that number are returned from the solution queue. If the work queue is empty and all worker threads are idling, then all the solutions in the queue are returned and the program exits. Note that the program may continue (co)recursively for indefinitely long.

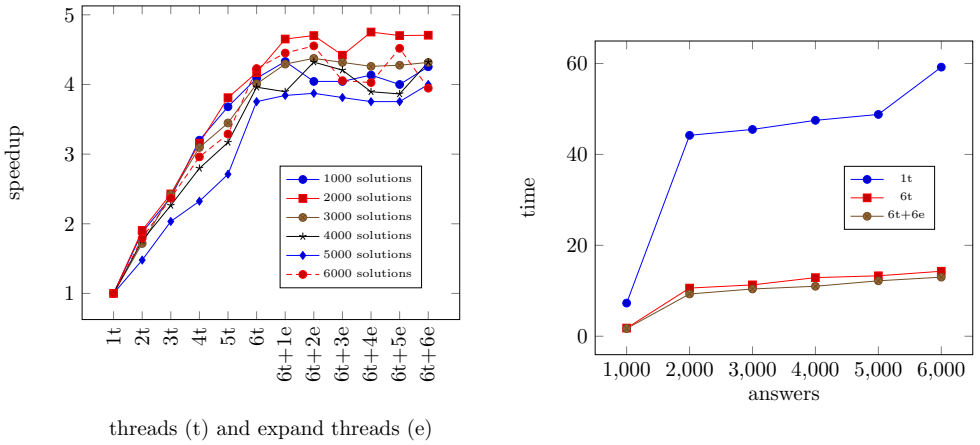

Fig. 15. **Left.** Speedup of the derivation of answers for the query `btree(X)`, relative to the base case with 1 worker thread, in the `BinaryTree` program for different parameters. **Right.** Comparison of the time needed to produce answers with different worker threads.

A PROLOG-like system with deterministic depth-search would produce solutions `btree(empty)`, `btree(tree(empty,0,empty))`, `btree(tree(empty,0,tree(empty,0,empty)))`, but not e.g. `btree(tree(empty,1,empty))`. Thereby, it does not generate the same set of solutions even if run indefinitely, and does not discover some of the solutions that CoALP does for the `BinaryTree` program.

5.3 Case Study: Binary Tree

Let us show the benefits of using parallelism to derive answers for the `BinaryTree` program. We focus on deriving different possibilities for substitution in the variable `X` in the query `btree(X)`.

Figure 15 shows the speedup when computing different amount of answers for the query `btree(X)` using different parameters. First of all, we can notice the great benefit of increasing the number of threads to derive answers when the parallel expansion option is not used; in particular, the maximum speedup is 4.23. The reason for these good results is simple: the main execution distributes the derivation of different solutions across all the available threads; and as each derivation is independent from the others, they can be run simultaneously. On the contrary, the number of expand threads that are used does not impact the performance of our implementation (the maximum speedup is 4.55 that is not too different from the maximum speedup obtained without expand threads).

Note also the considerable difference in the runtime that is necessary to generate 2000 solutions instead of 1000 solutions – a situation that does not happen in the rest of the cases. This is due to the fact that the generation of 1000 solutions requires at most 6 derivation steps; on the contrary, the generation of 2000 solutions requires, in some cases, 7 steps. Table 1 shows that there is a considerable difference to generate all the solutions requiring at most n derivation steps ($2^{n-1}\frac{(2n-2)!}{n!(n-1)!}$ and

	$\sum_{i=1}^{n} 2^{n-1} \frac{(2n-2)!}{n!(n-1)!}$ solutions	$1 + \sum_{i=1}^{n} 2^{n-1} \frac{(2n-2)!}{n!(n-1)!}$ solutions
$n = 2$	28ms	111ms
$n = 3$	178ms	635ms
$n = 4$	1.18s	4.68s
$n = 5$	8.92s	38.83s

Table 1
Difference of time when generating all the solutions that require at most n steps and when generating all the solutions requiring at most n derivation steps plus one solution that requires $n + 1$ steps.

$\sum_{i=1}^{n} 2^{n-1} \frac{(2n-2)!}{n!(n-1)!}$ are the number of solutions that require respectively n and at most n derivation steps) and the generation of all the solutions requiring at most n derivation steps plus one solution that requires $n + 1$ steps – the time increases up to 4 times.

As a final remark, we can compare the runtime that we obtain here with the Datalog results in Subsection 3.4. First-order implementation derives answers faster using the `BinaryTree` program than ground implementation does using the `BTG` programs. E.g. the first-order derivation method with the best configuration obtains 2000 solutions in $9s$ whereas it takes at least $65s$ in the `BTG` case. The lack of variables and references in the ground case means a more intense processing and use of memory space. The first-order version has fewer rules than the ground case, which makes the use of CPU caches more efficient.

This result is a good indication that the fibrational coalgebraic approach to first-order parallelism can be viable and efficient.

6 Conclusions

We have presented a novel implementation of CoALP, featuring two levels of parallelism: on the level of coinductive trees and on the level of coinductive derivations. We have traced carefully how the concepts undergo their transformation, from abstract coalgebraic semantics to logic algorithm and then to implementation. By doing this, we have exposed the constrictive component of the coalgebraic semantics [16, 17]. The concepts of coinductive trees and coinductive derivations arising from the semantics allow for many approaches of exploiting parallelism. These have been shown to provide practical speedup in our experimental implementation. Many improvements are planned to the current implementation, in particular, fine-tuning and-parallel tree transitions and memorisation. In the current stage of implementation, no memorisation of tree derivations is done. Much like in PROLOG systems, this however would save doing the same work multiple times by different workers. A first step to minimise this would be to share subtree structures created during and-parallel derivations. Also, expansion of goals can be cached for each clause making clause-tree templates not clause, but goal specific. Furthermore, *mgu's* from open nodes could be processed all at once instead of one at a time to reduce created copies of trees by finding compatible *mgu's* and applying them at same time. More sophisticated tabling algorithms to memorise computations for each clause can also

be added. Another project is to apply CoALP to type inference, similar to [2].

References

[1] G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.

[2] D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *TYPES'09*, volume 5497 of *LNCS*, pages 1–18, 2009.

[3] F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.

[4] F. Bonchi and F. Zanasi. Saturated semantics for coalgebraic logic programming. In *CALCO'13*, volume 8089 of *LNCS*, pages 80–94, 2013.

[5] R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *Theory Pract. Log. Program.*, 1(6):647–690, 2001.

[6] M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Inf. Comput.*, 169(1):23–80, 2001.

[7] C. Dwork, P.C. Kanellakis, and J.C. Mitchell. On the sequential nature of unification. *J. Logic Prog.*, 1:35–50, 1984.

[8] M. Gabrielli, G. Levi, and M.C. Meo. Observable behaviors and equivalences of logic programs. *Inf. Comput.*, 122(1):1–29, 1995.

[9] G. Gupta and V.S. Costa. Optimal implementation of and-or parallel PROLOG. In *PARLE'92*, pages 71–92, 1994.

[10] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of PROLOG Programs: a Survey. *ACM Trans. Comput. Log.*, 23:1–126, 2012.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[12] P. C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Prog.*, pages 547–585. Morgan Kaufmann, 1988.

[13] Y. Kinoshita and J. Power. A fibrational semantics for logic programs. In *Proc. Int. Workshop on Extensions of Logic Programming*, volume 1050 of *LNAI*, pages 177–191, 1996.

[14] E. Komendantskaya, G. McCusker, and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *AMAST'10*, volume 6486 of *LNCS*, pages 111–127, 2010.

[15] E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL'11*, LIPIcs, pages 352–366. Schloss Dagstuhl, 2011.

[16] E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO'11*, volume 6859 of *LNCS*, pages 268–282. Spinger, 2011.

[17] E. Komendantskaya, J. Power, and M. Schmidt. Coalgebraic logic programming. In *Journal of Logic and Computation*, 2014.

[18] E. Komendantskaya, M. Schmidt, and J. Heras. CoALP webpage: software and supporting documentation, 2013. http://www.computing.dundee.ac.uk/staff/katya/CoALP/.

[19] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[20] E. Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In *Euro-Par'95*, volume 966 of *LNCS*, pages 43–54, 1995.

[21] M. Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley, 2012.

[22] J. D. Ullman and A. Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

A Generation of balanced and unbalanced trees

```
BTA
Input: Number of iterations n

bits = {0,1}
trees = {tree(empty,0,empty), tree(empty,1,empty)}
new_trees = {}
result = {empty,tree(empty,0,empty), tree(empty,1,empty)}
i = 0

while (i < n)
  for each t1 in trees
     for each t2 in trees
        for each b in bits
            new_trees = new_trees ∪ {tree(t1,b,t2)}
        end for
    end for
  end for
  i = i + 1
  result = result ∪ new_trees
  trees = new_trees
  new_trees={}
end while

return result
```

Fig. A.1. The BTA algorithm generating balanced trees.

```
UTA
Input: Number of iterations n

bits = {0,1}
trees = {empty, tree(empty,0,empty), tree(empty,1,empty)}
new_trees = {}
i = 0

while (i < n)
  for each t1 in trees
     for each t2 in trees
        for each b in bits
            new_trees = new_trees ∪ {tree(t1,b,t2)}
        end for
    end for
  end for
  i = i + 1
  trees = trees ∪ new_trees
  new_trees={}
end while

return trees
```

Fig. A.2. The UTA algorithm generating unbalanced trees.